PROMISES AND ASYNCHRONOUS CODE

Excerpt from:
Learning Web Design, 6e

by Jennifer Robbins
with contributions by Aaron Gustafson

by Aaron Gustafson

Author's Note: *This article supplements* **Chapter 25, Next-Level JavaScript** *in* Learning Web Design, 6e.

Synchronous code is expected to execute immediately and each statement in your program is evaluated before the program moves on to the next statement. Consider this example:

```
let volume = 10;
volume++;
console.log( volume ); // 11
```

Here, the first statement declares the `volume` variable and assigns it a value of 10. The second statement takes that value and increments it by 1, making the value of `volume` 11. The third statement logs the value of `volume` to the console.

This approach is fine for operations like this that execute quickly, but sometimes the task you need to do will take an indeterminate amount of time to complete. It might be an extensive math calculation that takes a while to run. You might need to get contents of a file on the web, where network conditions and the file's size will impact how long it takes to get the file. Or maybe you need to get some records from a database. These tasks take time and if you were to do them with synchronous code, it would make a browser unresponsive until the task was complete. That's not the kind of experience that's going to make any user happy.

To address this challenge, JavaScript added support for running code asynchronously (or "async"). **Asynchronous code** relies on a special kind of object called a `Promise`. A Promise type keeps tabs on whatever the expensive operation is, but it doesn't hold up the remainder of your code from executing.

NOTE

Variables and functions that I have created and named appear in `orange`. You can assume that keywords, operators, variables, and functions not in orage are part of the built-in functionality of JavaScript.

NOTE:

`setTimeout()` *takes two arguments. The first is the function you want to run. The second is the number of milliseconds you want to wait before running that function. In my case the function is an anonymous function that increments volume. I've set it to run after 1 second, which is 1000 milliseconds.*

To illustrate how it works, I'm going to use a built-in JavaScript method called **`setTimeout()`** to stand in for a long-running operation. In the code below, I've set it up so that the Promise will take 1 second to **resolve** (i.e., complete without error) and when it resolves, the value of `volume` will be incremented by 1. Then it will log the new value of `volume` to the console.

```
let volume = 10;
new Promise(function( resolve ){
  setTimeout(function() {
    resolve(volume++);
  }, 1000);
}).then(function(){
  console.log( volume ); // 11
});
```

Here I've replaced the statement that increments the value of `volume` with a statement that creates a new **`Promise`** using the **`Promise()` constructor**. A constructor is what you use to create new instances of a particular object type, in this case a **`Promise`** object. Whenever you see the **`new`** keyword, it signals you're using a constructor.

The value you pass into a **`Promise`** constructor is a function. That function will receive two arguments from the Promise. The first is a function you can use to resolve the Promise. The second is a function you can use to **reject** the Promise (which is what we call it when the code in the Promise fails). You could technically name these arguments whatever you want — it's your function to define — but developers typically call them `resolve()` and `reject()` for clarity.

Within the function I've defined is the delayed — using **`setTimeout()`** — increment statement. I could have incremented volume in one statement and then called resolve in a separate statement, but you can also pass an argument to `resolve()`, so I chose to do it all in a single statement.

To log the correct value to the console, we need to wait for the Promise to resolve. Thankfully, a **`Promise`** has a **`then()`** method which will be called when that happens. To prepare code for that, you call **`then()`** and pass it a function as an argument and that function will be run when the Promise resolves. In this example, that's where I log the value of volume to the console.

If I copied all this code, pasted it into the JavaScript console in my browser, and ran it, nothing would happen for a second. Then, as if by magic, the value 11 would get logged to the console (see **FIGURE A**).

This was an incredibly contrived example, but it illustrates how we can begin thinking about JavaScript running on multiple, independent timelines instead of one. It also provides a path for us to improve our users' experiences by not locking up their browser when we need to do more intensive JavaScript operations like fetching content from other servers or querying databases.

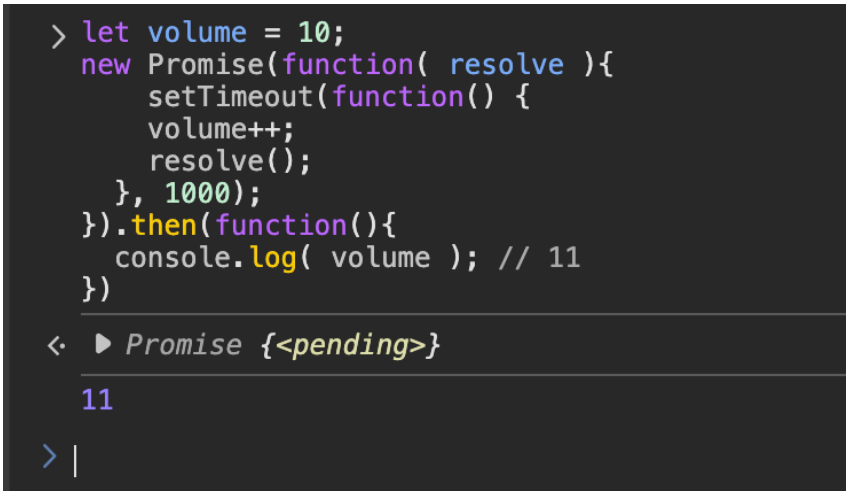

FIGURE A. In this screenshot from the console, I've pasted and run the Promise example, which resolves to a pending Promise. Then, after a second, the value 11 appears in the log above the prompt.

As you begin doing more advanced JavaScript work, you'll encounter Promises all over the place. Service Workers make extensive use of them as do JavaScript frameworks. For more on how they work and some excellent exercises that walk you through using them, check out "What is a Promise? JavaScript Promises for Beginners" by Kingsley Ubah (*www.freecodecamp.org/news/what-is-promise-in-javascript-for-beginners/*).